Deep generative models
Part 1: Generator networks

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

In this lecture, we'll look at generative modeling, The business of training probability models that are too complex to give us an explicit density function over our feature space, but that do allow us to sample points. If we train them well, we get points that look like those in our dataset.

---

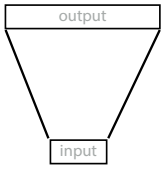
generative models

Here is the example, we gave in the first lecture. A deep neural network from which we can sample highly realistic images of human faces.

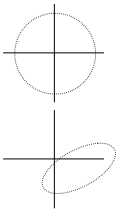
source: **A Style-Based Generator Architecture for Generative Adversarial Networks**, Karras et al.

---

visual shorthand



any neural network



a (standard)
multivariate normal distribution

3

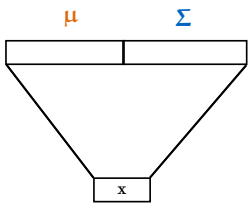
In the rest of the lecture, we will use the following visual shorthand. The diagram on the left represents any kind of neural network. We don';t care about the precise architecture, whether it has one or a hundred hidden layers and whether it uses fully connected layers of convolutions, we just care about the shape of the input and the output.

The image on the right represents a multivariate normal distribution. Think of this as a contour line for the density function. We've drawn it in a 2D space, but we'll use it as a representation for MVNs in higher dimensional spaces as well. If the MVN is nonstandard, we'll draw it as an ellipse somewhere else in space.

---

how to turn a NN into a probability distribution

option 1:



$$p(\mathbf{y} \mid \mathbf{x}) = N(\mathbf{y} \mid \mu, \Sigma) \quad \text{with } (\mu, \Sigma) = f(\mathbf{x})$$

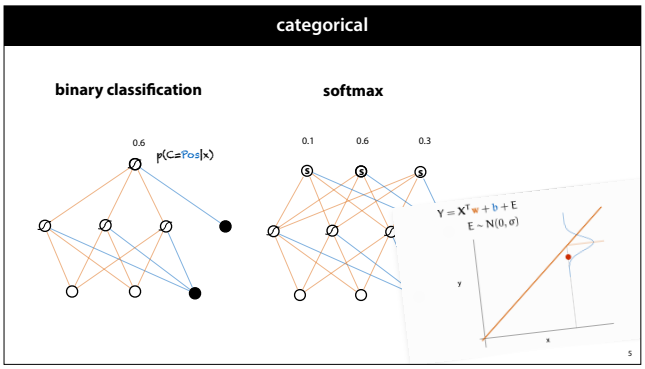
4

A plain neural network is purely deterministic. It translates an input to an output and does the same thing every time with no randomness. How to we turn this into a probability distribution?

The first option is to take the output and to interpret it as the parameters of a multivariate normal (a μ and a Σ). If the dimensionality of the output is large, we'll often take Σ to be a diagonal matrix (which is why it's the same size as μ in this image).

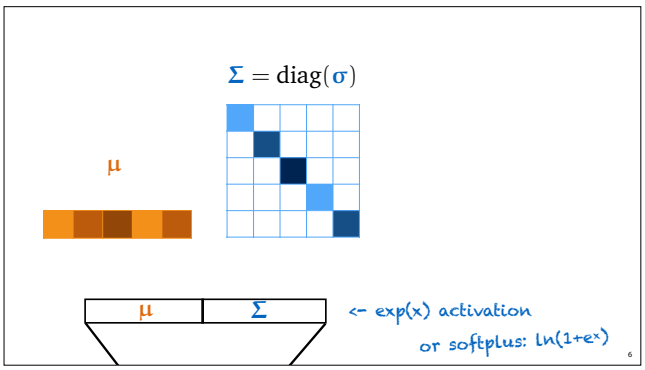
The distribution we represent in this way is always a *conditional* distribution. We get a probability on our output space, conditioned on the input value $\mathbf{x}$. If we change $\mathbf{x}$, we get a different distribution on the output

space.



This principle is not new. In neural network classification, the standard approaches we've seen (a linear regression layer and a softmax output) the neural network also produces the parameters for a probability distribution on the output space (a Bernoulli distribution on the left and a Categorical on the right).

Even linear regression, as we saw in the previous lecture, can be thought of as providing the mean for an error distribution, and in that case the maximum likelihood objective is equivalent to training by squared errors. We just don't provide the variance of the output distribution in that case, only the mean.

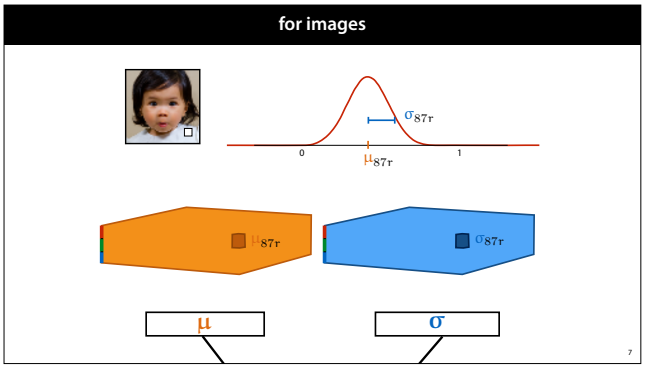


If we do provide the mean and the variance of an output distribution, it looks like this (for an n-dimensional output space). We simply split the output layer in two parts, and interpret one part as the mean and the other as the covariance matrix.

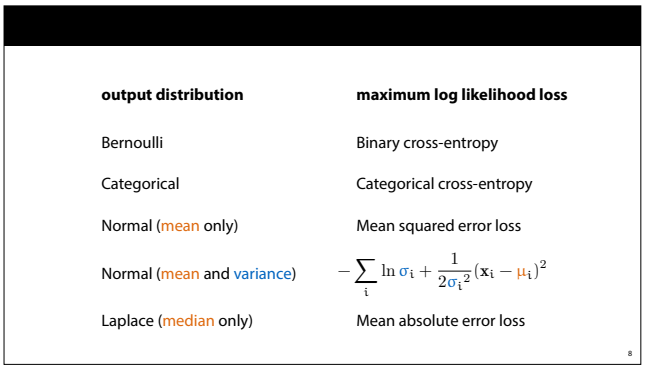
Since representing a full covariance matrix would grow very big, we usually assume that the covariance matrix only has nonzero values on the diagonal. That way the representation of the covariance requires as many arguments as the representations of the mean, and we can simply split the output into two halves.

Equivalently, we can think of the output distribution as putting an independent 1D Gaussian on each dimension, with a mean and variance provided for each.

For the mean, we can use a linear activation, since it can have any value, including negative values. However, the values of the covariance matrix need to be positive. To achieve this, we often exponentiate them. We'll call this an **exponential activation**. An alternative option is the **softplus** function $\ln(1 + e^x)$, which grows less explosively.

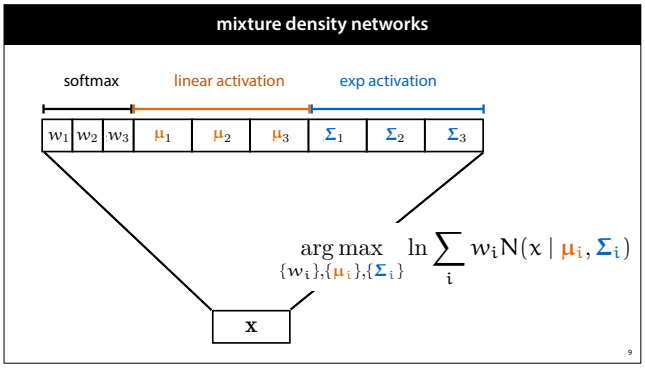


Here's what that looks like when we generate an image. The output distribution give us a mean value for every channel of every pixel (a 3-tensor) and a corresponding variance for every mean. If we look at what that tells us about the red value of the pixel at coordinate (8, 7) we see that we get a univariate Gaussian with a particular mean and a variance. The mean tells us the network's best guess for that value, and the variance tells us how *certain* the network is about this output. We should note that neural networks by default are terrible at estimating how certain they should be, so we should take this value with a grain of salt.



For many output distributions, maximizing the log likelihood of our data leads to loss functions that we've seen already.

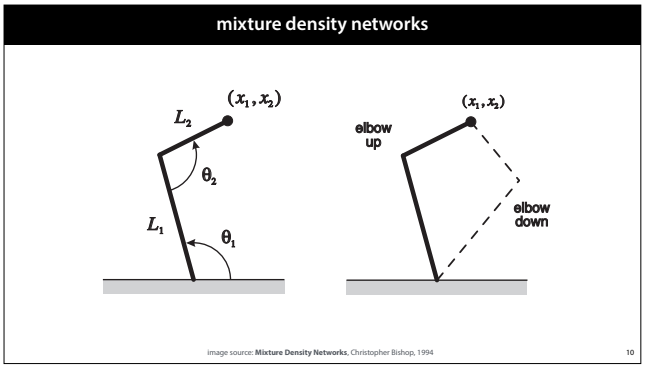
The loss function for an normal output distribution with a mean and variance, is a modified squared error. We can set the variance larger to reduce the impact of the squared errors, but we pay a penalty of the logarithm of sigma. If we know we are going to get the output value for instance i exactly right, then we will get almost no squared error and we can set the variance very small, paying little penalty. If we we're less sure, then we expect a sizable squared error and we should increase the variance to reduce the impact a little.



If we want to go all out, we can even make our neural network output the parameters of a Gaussian mixture. This is called a **mixture density network**.

All we need to do is make sure that we have one output node for every parameter required, and apply different activations to the different kinds of parameters. The means get a linear activation and the covariances get an exponential activation as before. The weights need to sum to one, so we need to give them a softmax activation (over just these three weights).

If we want to train with maximum likelihood, we encounter this sum-inside-a-logarithm function again, which is difficult to deal with. But this time, it's not such a headache. As we noted in the last lecture we can work out the gradient for this loss, it's just a very ugly function. Since we are using backpropagation anyway, that needn't worry us here. All we need to work out ad the local derivatives for functions like the logarithm and the sum, and those are usually provided by systems like Pytorch and Keras anyway.

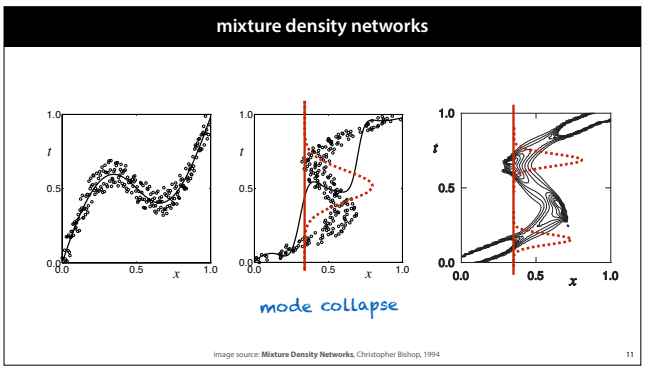


The mixture density network may seem like overkill, but it's actually very useful in cases where we have multiple valid answers.

Consider the problem of inverse kinematics in robotics. We have a robot arm with two joints, and we know where in space we want the hand of the arm to be. What angles should we set the two joints to? This is a great application for machine learning: it's a relatively simple, smooth function. It's easy to generate examples, and explicit solutions are a pain to write, and not robust to noise in the control of the robot. So we can solve it with a neural net.

The inputs are the two coordinates where we want the hand to be (x_1, x_2), and the outputs are the two angles of the joints (θ_1, θ_2). The problem we run into, is that for every input, there are two solutions. One with the elbow up, and one with the elbow down. A normal neural network trained with an MSE loss would not pick one or the other, but it would *average between the two*.

A mixture density network with two components can fix this problem. For each input, it can simply put its components on the two outputs



The problem with the robot arm is that the task is uniquely determined in one direction—every configuration of the robot arms leads to a unique point in space for the hand—but not when we try to reason backward from the hand to the configuration of the joints.

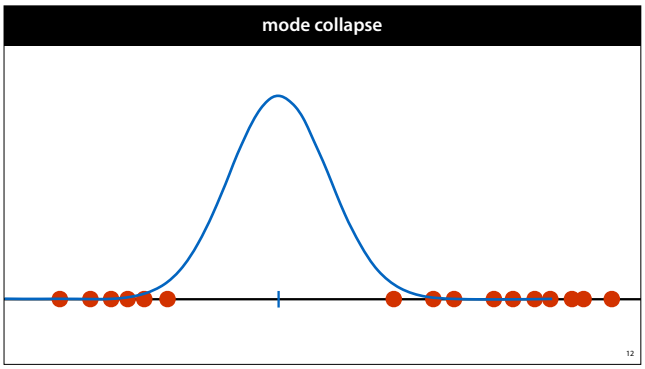
This often happens when you take a trivial regression problem and flip around the inputs and the outputs. On the left, we see a very simple regression problem. for every input x, there is a unique output t, if with a little noise. A small network easily finds a nice fit. If, however, we flip around the values of t and x, the problem becomes really difficult. For an input like 0.4, there are two distinct regions that both have a high density of points.

The network can only predict one output value, so it ends up averaging between the two, putting the output in a part of space where there are no data at all. If we are training with mean squared error loss, we can this of the output as fitting he mean of a normal distribution to the output. We can give the neural network control over the variance of this distribution as well, but all that achieves is that the variance grows to cover both groups of points. The mean stays in the same place.

By contrast, the mixture density network can output a distribution with two peaks, known as its **modes**. This allows it to cover the two groups of points in the

output, and so solve the problem in a much more useful way

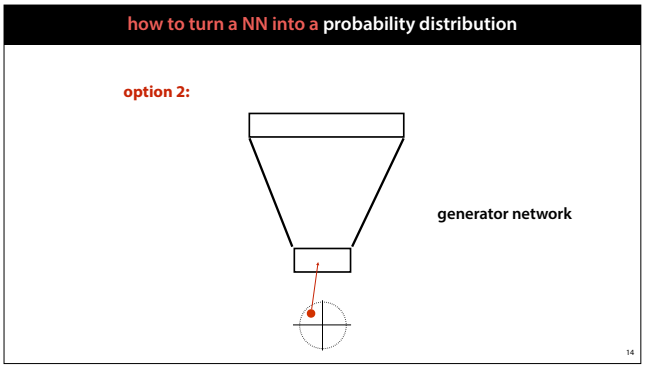
The general problem in the middle picture is called **mode collapse**.



If our data is spread out in space in a complex, clustered pattern, and fit a simple unimodel distribution to it, that is, a distribution with one peak. The result is a distribution that puts all the probability mass on the average of our points, but very little probability mass where the points actually are.

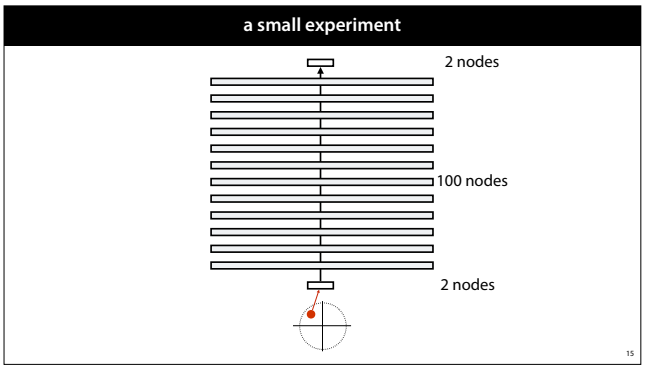
Mixture density networks go some way towards letting us capture more complex distributions in our neural networks, but when we want to capture something as complex as the distribution on images representing human faces, they're still insufficient.

A mixture model with k components gives us k modes. So in the space of images, we can pick k images to give high probability and the rest is just a simple gaussian shape around those k points. The distribution on human faces has infinitely many modes in this space that should all be equally likely. To achieve a distribution this complex, we need to use the power of the neural net, not just to choose a finite set of modes, but to control the whole shape of the probability function.



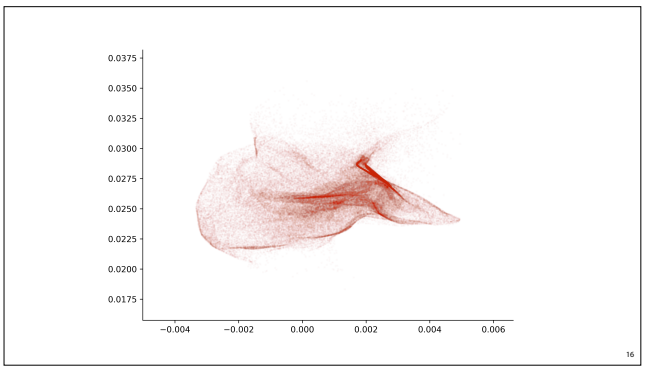
Here's a more powerful approach: we put the normal distribution at the *start* of the network instead of at the end of it. We sample a point from some straightforward distribution, usually a standard normal distribution, and we feed that point to a neural net. The result of these two steps is a random point, so we've defined another probability distribution. We call this construction a **generator network**.

Compare this to how we defined parametrized multivariate normals in the previous lecture: we started with a standard normal distribution, and we applied a linear transformation. This is the same thing, but we've replaced the linear transformation by a nonlinear one.



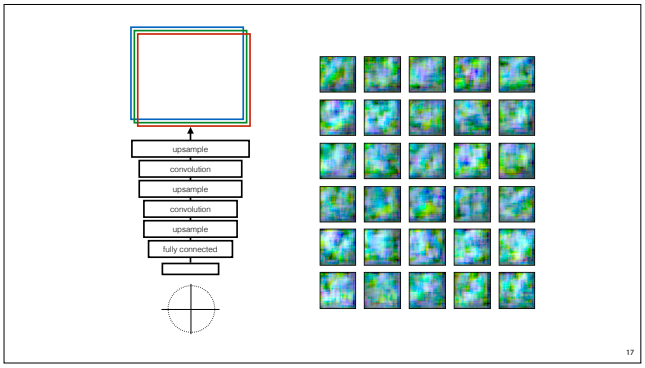
To see what kind of distributions we might get when we do this, let's try a little experiment.

We wire up a random network as shown: a two-node input layer, followed by 12, 100-node fully-connected hidden layers with ReLU activations., and a final transformation back to two points. We *don't train* the network. We just use Glorot initialisation to pick the parameters, and then sample some points. Since the output is 2D, we can easily scatter-plot it.



Here's a plot of 100k points sampled in this way. Clearly, we've define a highly complex distribution. Instead of having a finite set of single points as modes, we get strands of high probability in space, and sheets of lower, but nonzero probability. Remember, this network hasn't been trained, so it's not representing anything meaningful, but it shows that the kinds of distributions we can represent in this way is a highly complex family.

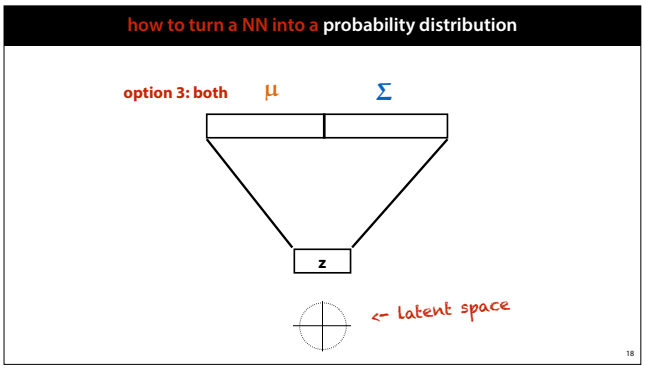
NB: Note that the variance has shrunk, and the mean has drifted away from (0, 0); apparently the weight initialisation is not quite perfect.



We can also use this trick to generate images. A normal convolutional net starts with a low-channel, high resolution image, and slowly decreases the resolution by maxpooling, while increasing the number of channels. Here, we reverse the process. We shape our input into a low resolution image with a large number of channels. We slowly increase the resolution by upsampling layers, and deacrease the number of channels.

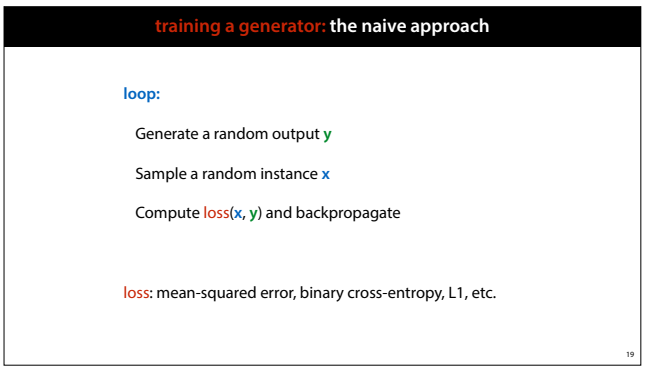
We can use regular convolution layers, or deconvolutions, which are convolutions where one patch in the output is wired to a single node in the input. both approaches give us effective generator networks for images.

NB: I've enhanced the contrast and saturation to ensure that the colors show up on a projector.



Of course, we can also do both: we sample the input from a standard MVN, interpret the output as another MVN, and then sample from that.

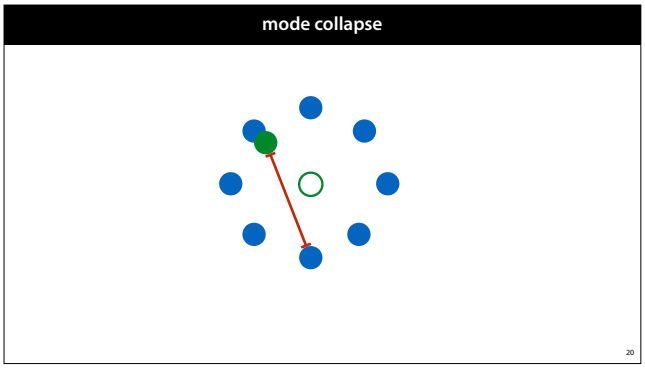
In these kinds of generator networks, the input is often called **z**, and the space of inputs is often called the **latent space**.



So the big question is, how do we train a generator network? Given some data, how do we set the weights of the network so that the output starts to look like our examples?

We'll start with what doesn't work. Here is a naive approach: we simply sample a random point (e.g. a picture) from the data, and sample a point from the model and train on how close they are.

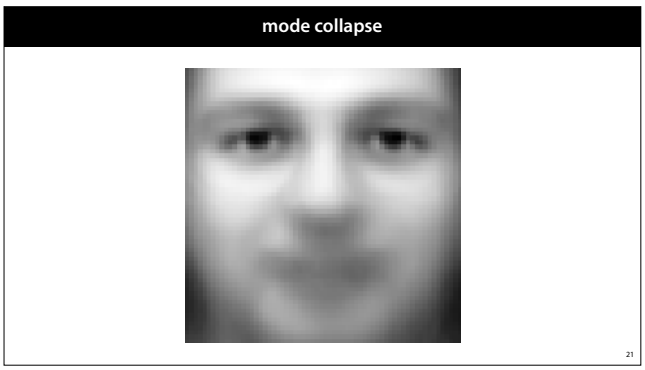
The loss could be any distance function between two tensors. The mean-squared error over the elements is a simple approach. If the elements are values between 0 and 1 (like in images), the binary cross-entropy makes sense too. For images the absolute value of the error (also called L1 loss) is also popular.



If we implement the naive approach, we get something called *mode collapse*. Here is an example: the blue points represent the modes (likely points) of the data. The green point is generated by the model. It's close to one of the blue points, so the model should be rewarded, but it's also far away from some of the other points. During training, there's no guarantee that we pair it up with the correct point, and we are likely to compute the loss to a completely different point.

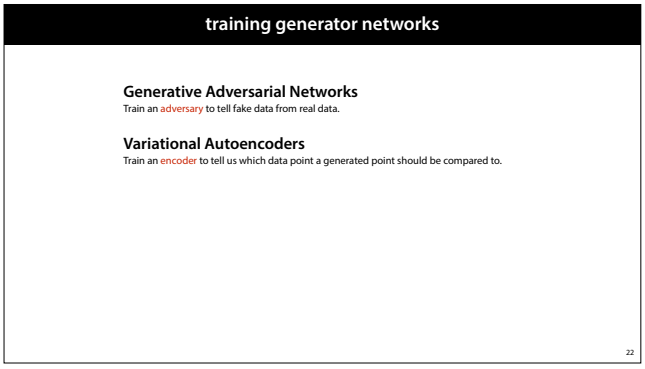
On average the model is punished for generating such points as much as it is rewarded. The model that generates only the open point in the middle gets the same loss (and less variance). Under backpropagation, neural networks tend to converge to a distribution that generates only the open point over and over again.

In other words, the many different modes (areas of high probability) of the data distribution end up being averaged ("collapsing") into a single point.

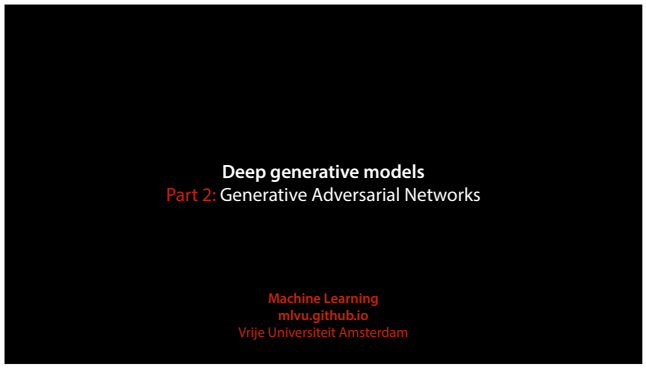


So we're back to mode collapse. Even though we have a probability distribution that is able to represent highly complex, multi-modal outputs, if we train it like this, we still end up producing a unimodal output centered on the mean of our data.

How do we get the the network to *imagine details*, instead of averageing over all possibilities?

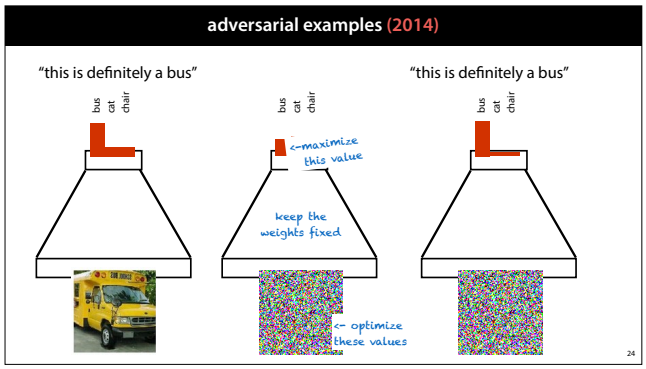


There are two main approaches, which we'll discuss in the rest of the lecture.



In the last video, we defined generator networks, and we saw that they represent a very rich family of probability distributions. We also saw, that training them can be a tricky business.

In this video we'll look one way of training such networks: the method of generative adversarial networks.

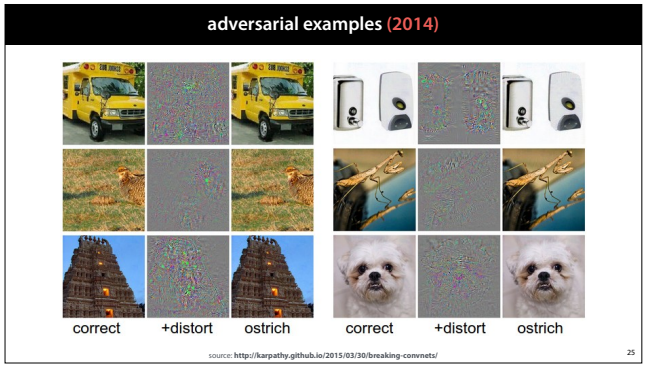


GANs originated just after ConvNets were breaking new ground, showing spectacular, sometimes super-human, performance in image labeling. The suggestion aroe that conv nets were doing more or less as the same as what humans do when they look at something.

To verify this, researchers decided to start investigating what kind of inputs would make a trained ConvNet give a certain output. This is easy to do, you just compute the gradient with respect to the input of the network, and train the input to maximise the activation of a particular label. This is similar to the feature visualizing approach we saw before, but you do it on the output nodes instead of the hidden nodes.

You would expect that if you start with a random image, and follow the gradient to maximize the activation of the output node corresponding to the label "bus", you'd get a picture of a bus. Or at least something that looks a little bit like a bus. What you actually get is something that is indistinguishable from the noise you started with. Only the very tiniest of changes is require to make the network see a bus.

These are called **adversarial examples**. instances that are specifically crafted to trip up a given model.



The researchers also found that if they started the search not at a random image, but at an image of another class, all that was needed to turn it into another class was a very small distortion. So small, that to us the image looked unchanged. In short, a tiny bit of distortion is enough to make a CNN think that a picture of a bus is a picture of an ostrich.

Adversarial examples are an active area of research (both how to generate them and make models more robust against them).

source: Eykholt, Kevin, et al. "Robust physical-world attacks on deep learning visual classification." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018.

26

Even manipulating objects in the physical world can have this effect. A stop sign can be made to look like a different traffic sign by the simple addition of some stickers.

generative adversarial networks

loop:

train a classifier to tell X_{Pos} from X_{Neg}

Generate adversarial examples
Clearly not Pos, but the classifier thinks so anyway

Add the adversarial examples to X_{Neg}

The classifier (discriminator) gets more *robust*, the generator gets more *realistic*.

27

Pretty soon, this bad news was turned into good news by realising that if you can generate adversarial examples automatically, you can also add them to the dataset *as negatives* and retrain your network to make it more robust. You can simply tell your network that these things are not stop signs, and to learn again.

We can think of this as a kind of iterated 2 player game (or an arms race). The Generator (the process that generates the adversarial examples) tries to get good enough to fool the classifier and the classifier tries to get good enough to tell the fakes from the true examples.

This is the basic idea of the **generative adversarial**

GANs

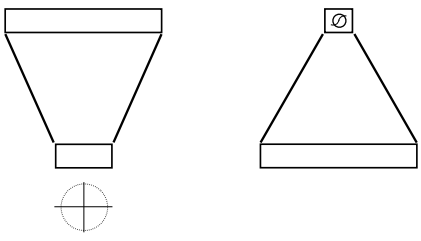
- Vanilla GANs
- Conditional GANs
- CycleGAN
- StyleGAN

28

We'll look at four different examples of GANs. We'll call the basic approach the "vanilla GAN"

end-to-end GANs

generator: G **discriminator: D**



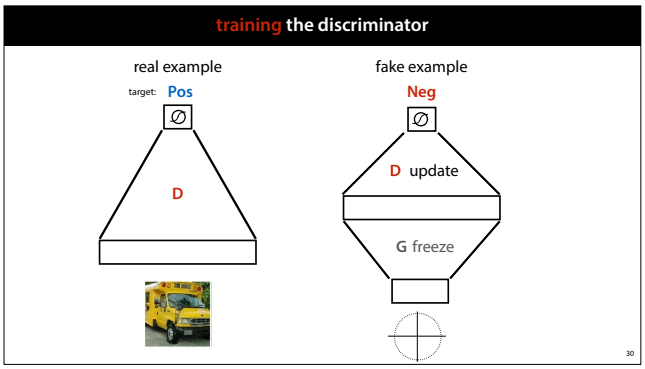
29

Generating adversarial examples by gradient descent is possible, but it's much nicer if both our generator and our discriminator are separate neural networks. This will lead to a much cleaner approach for training GANs.

We will draw the two components like this. The **generator** takes an input sampled from a standard MVN and produces an image. This is a generator network are described in the previous video (with no ouptu distribution). The **discriminator** takes an image and classifies it as Pos (a real image) or Neg (a fake image sampled from the generator).

If we have other images that are not of the target class, we can add those to the negative examples as well, but

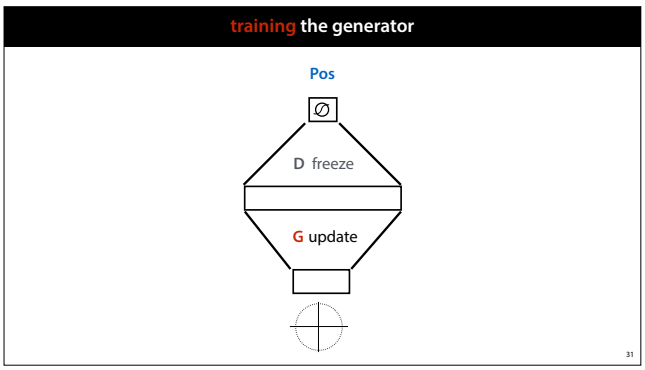
often, the positive class is just our dataset (like a collection of human faces), and the negative class is just the fake images created by the generator.



To train the discriminator, we feed it examples from the positive class, and train it to classify these as Pos.

We also sample images from the generator (whose weights we keep fixed) and train the discriminator to make these negative. At first these will just be random noise, but there's little harm in telling our network that such images are not busses (or whatever our positive class is).

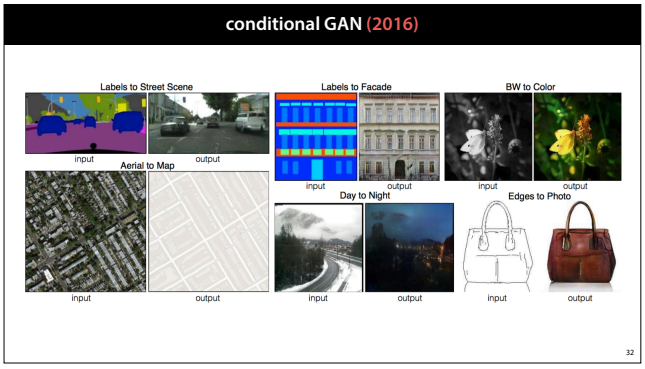
Note that since the generator is a neural network, we don't need to collect a dataset of fake images. We can just stick the discriminator on top of the generator, and train it, by gradient descent to classify the result are negative. We just need to make sure to freeze the



Then, to train the generator, we freeze the discriminator and train the weights of the generator to produce images that cause the discriminator to label them as **Positive**.

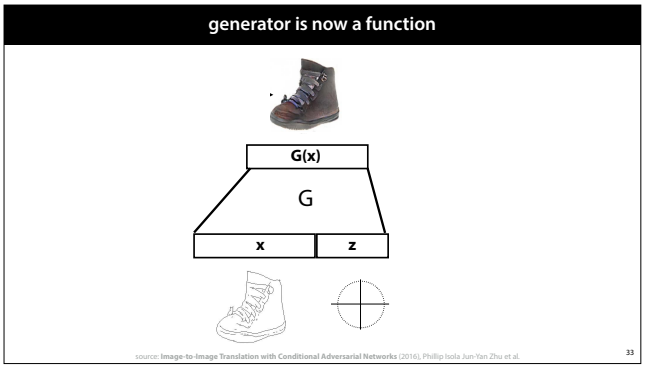
We don't need to wait for either step to converge. We can just train a the discriminator for one batch (i.e. one step of gradient descent) and then train the generator for one batch, and so on.

And this is what we'll call the **vanilla GAN**.

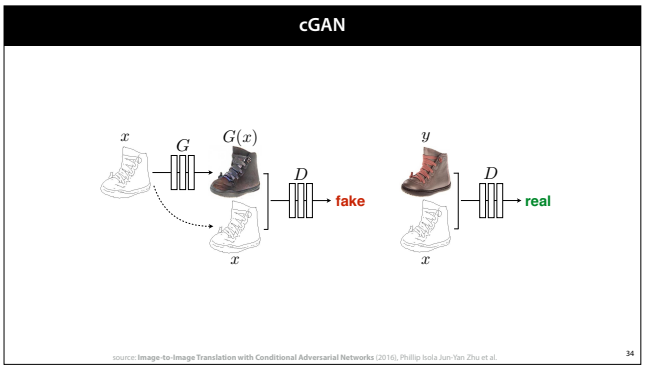


Sometimes we want to train the network to take an input, but to generate the output probabilistically. For instance, when we train a network to color in a black-and-white photograph of a flower, it could choose many colors for the flower. We want to avoid mode collapse here: instead of averaging over all possible colors, giving us a brown or gray flower, we want it to pick one color, from all the possibilities.

A conditional GAN lets us train generator networks that can do this.



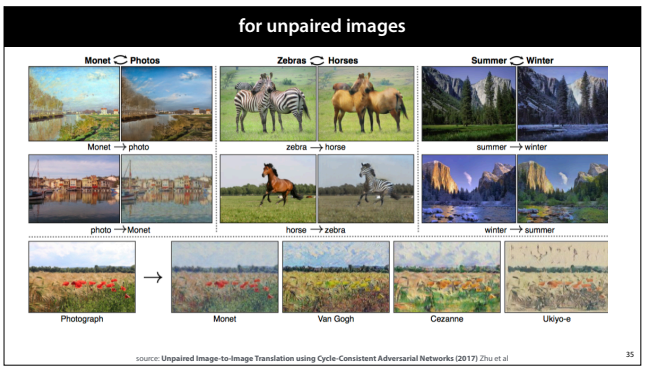
In a conditional GAN, the generator is a function with an image input, which it maps it to an image output. However, it uses randomness to imagine specific details in the output. running this generator twice would result in different shoes that are both "correct" instantiations of the input line drawing.



The discriminator takes *pairs of inputs and outputs*. If these come from the generator, they should be classified as fake (negative) and if they come from the data, they should be classified as real (positive).

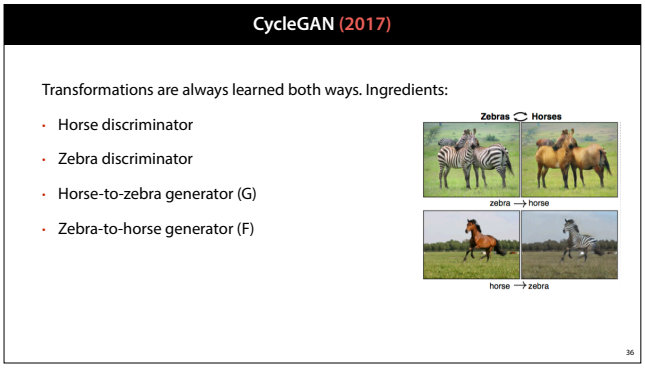
The generator is trained in two ways.

1. We freeze the weights of the discriminator, as before, and train the generator to produce thins that the discriminator will think are real.
2. We feed it and input from the data, and back propagate on the corresponding output (using L1 loss).



The conditional GAN works really well, but only if we have an example of a specific output for the input. For some tasks, we don't have paired images. We only have unmatched bags of images in two domains.

If we randomly match one image in one domain to an image in another domain, we get mode collapse again.

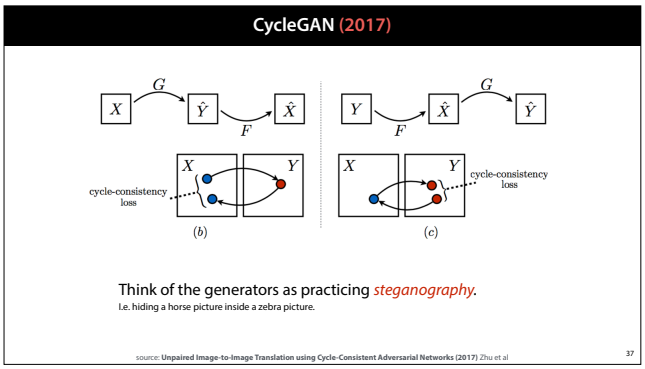


CycleGANs achieve this by adding a "cycle consistency term" to the loss function. For instance, in the horse-to-zebra example, if we transform a horse to a zebra and back again, the result should be close to the original image. Thus, the objective becomes to train a horse-to-zebra transformer *and* a zebra-to-horse transformer together in such a way that:

- a horse-discriminator can't tell the generated horses from real ones (and likewise for a zebra discriminator)
- the cycle consistency loss for both combined is low.

After some training of the generators, the discriminators are retrained with with the original

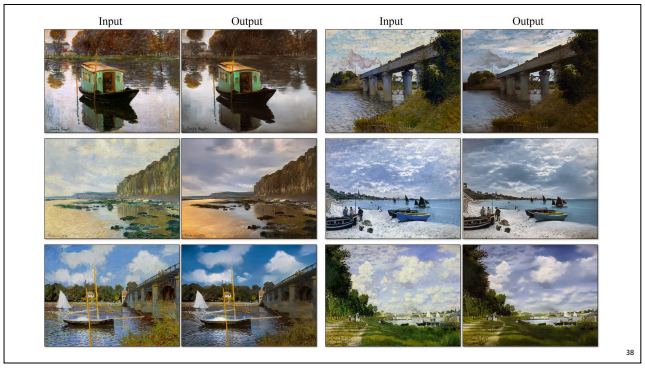
data and some generated horses and zebras.



CycleGANs achieve this by adding a "cycle consistency term" to the loss function. For instance, in the horse-to-zebra example, if we transform a horse to a zebra and back again, the result should be close to the original image. Thus, the objective becomes to train a horse-to-zebra transformer and a zebra-to-horse transformer together in such a way that:

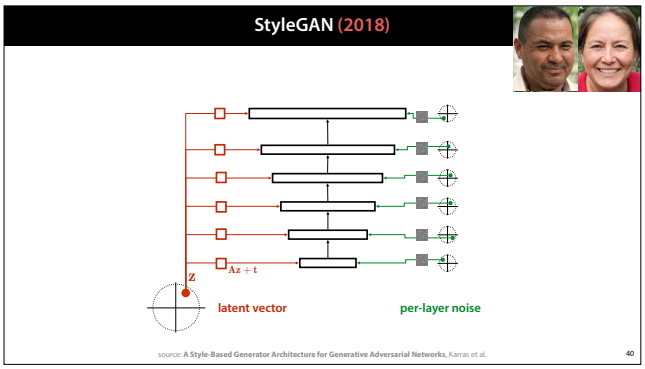
- a horse-discriminator can't tell the generated horses from real ones (and likewise for a zebra discriminator)
- the cycle consistency loss for both combined is low.

After some training of the generators, the discriminators are retrained with with the original data and some generated horses and zebras.



The CycleGAN works surprisingly well. Here's how it maps photographs to impressionist paintings and vice versa.

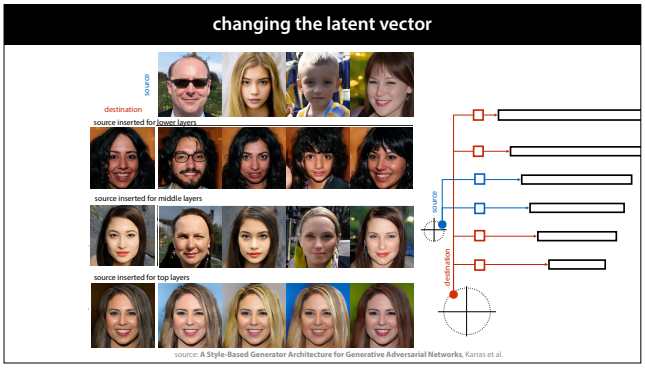
It doesn't always work perfectly, though.



Finally, let's take a look at the **StyleGAN**, the network that generated the faces we first saw in the introduction. This is basically a Vanilla GAN, with all most of the special tricks in the way the generator is constructed. It uses too many tricks to discuss here in detail, so we'll just focus on one aspect: the idea that the latent vector is fed to the network at each stage of its forward pass.

Since an image generator starts with a coarse (low resolution), high level description of an image, and slowly fills in the details, feeding it the latent vector at every layer (transformed by an affine transformation to fit it to the shape of the data at that stage), allows it to use different parts of the latent vector to describe different aspects of the image (the authors call these "styles").

The network also receives separate extra random noise per layer, that allows it to make random choices. Without this, all randomness would have to come from the latent vector.



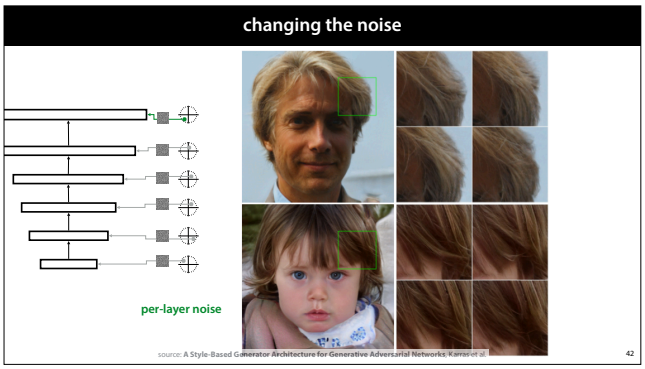
To see how this works, we can try to manipulate the network, by changing the latent vector to another for some of the layers. In this example all images on the margins are people that are generated for a particular single latent vector.

We then re-generate the image for the destination, except that for a few layers (at the bottom, middle or top), we use the source latent vector instead.

As we see, overriding the bottom layers changes things like gender, age and hair length, but not ethnicity. For the middle layer, the age is largely taken from the destination image, but the ethnicity is now override by the source. Finally for the top layers, only surface

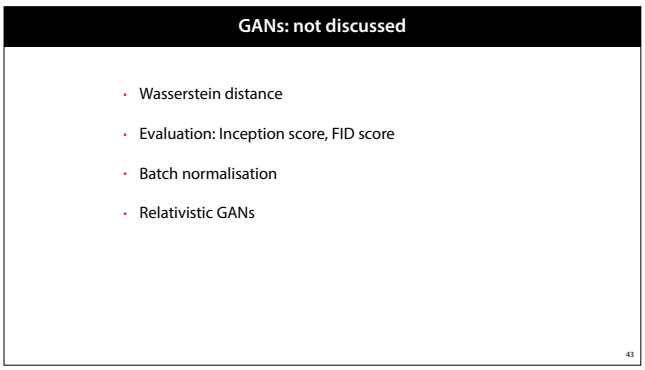
details are changed.

This kind of manipulation was done during training as well, to ensure that it would lead to faces that fool the discriminator.



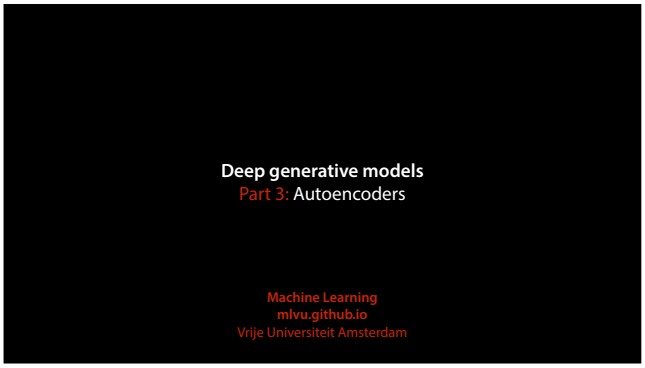
Let's look at the other side of the network: the noise inputs.

If we keep all the latent and noise inputs the same except for the very last noise input, we can see what the noise achieves: the man's hair is equally messy in each generated example, but exactly in what way it's messy changes per sample. The network uses the noise to determine the precise orientation of the individual "hairs".



We've given you a high level overview of GANs, which will hopefully give you an intuitive grasp of how they work. However, GANs are notoriously difficult to train, and many other tricks are required to get them to work. Here are some phrases you should Google if you decide to try implementing your own GAN.

In the next video, we'll look at a completely different approach to training generator networks: autoencoders.



What can we do with a generator?

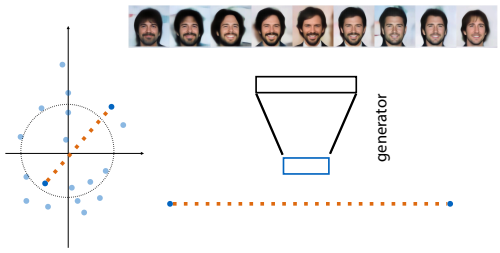
- Generate "new" data
- Interpolation
- Data manipulation
- Dimensionality reduction

45

In the last video, we saw our first strategy for training generator networks. What can we do once we have a well-trained generator network? We'll look at four use cases.

The first, of course is that we can generate data that looks like it came from the same distribution as ours.

interpolation



source: **Sampling Generative Networks**, Tom White

46

If we take two points in the latent space, and draw a line between them, we can pick evenly spaced points on that line and decode them. If the generator is good, this should give us a smooth transition from one point to the other, and each point should result in a convincing example of our output domain.

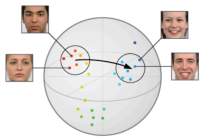
interpolation grid

source: **Sampling Generative Networks**, Tom White

47

We can also draw an interpolation grid; we just map the corners of a square lattice of equally spaced points to four points in our latent space.

spherical linear interpolation



source: **Sampling Generative Networks**, Tom White

48

If the latent space is high dimensional, most of the probability of the standard MVN is near the edges of the radius-1 hypersphere (not in the centre as it is in 1, 2 and 3-dimensional MVNs). High-dimensional MVNs look more like a soap bubble than the dense pointcloud we're used to seeing in low-dimensional visualizations.

For that reason, we get better results if we interpolate along an arc instead of along a straight line. This is called **spherical linear interpolation**.

Interpolation experiments are a great way to help us get to grips with the structure of our latent space.

interpolation on real data

source: Abdal, Rameen, Yipeng Qin, and Peter Wonka. "Image2stylegan: How to embed images into the stylegan latent space?." *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019.

49

What if we want to interpolate between points in our dataset? It's possible to do this with a GAN trained generator, but to make this work, we first have to *find* our data points in the generator. Remember, during training the dsicriminator is the only network that gets to see the actual data. We never explicitly map the data to the latent space.

We can tack a mapping from data to latent space onto the network afetr training (as was done for these images), but we can also learn such a mapping directly. As it happens, this can help us to train the generator in a much more direct way.

What can we do with a generator?

- Generative modelling
- Interpolation
- Data manipulation
  autoencoders only
- Dimensionality reduction
  autoencoders only

requires mapping into the latent space

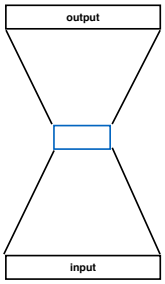
50

Note that such a mapping would also give us a **dimensionsality reduction**. We can see the latent space representation of the data as a reduced dimensionality representation of the input.

We'll focus on the prespective of dimensionality reduction for the rest of this video, to set up basic autoencoders. We can get a generator network out of these, but it's a bit of an afterthought. In the next video, we'll see how to train generator networks with a data-to-latent-space mapping in a more principled way.

---

autoencoders



bottleneck architecture for dimensionality reduction.

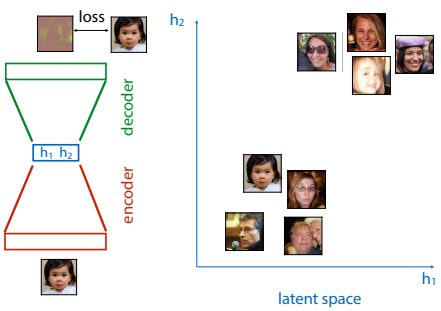
input should be as close as possible to the output

but: must pass through a small representation.

51

Here's what a simple **autoencoder** looks like. It's is a particular type of neural network, shaped like an hourglass. Its job is just to make the output as close to the input as possible, but somewhere in the network there is a small layer that functions as a bottleneck.

After the network is trained, this small layer becomes a compressed low-dimensional representation of the input.



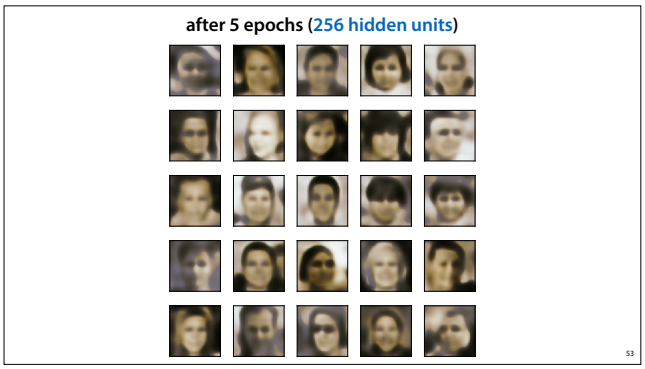
52

Here's the picture in detail. We call the bottom half of the network the encoder and the top half the decoder.

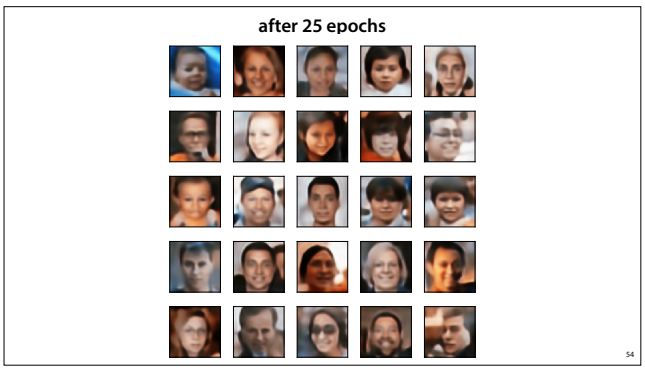
We call the blue layer the latent representation of the input. If train an autoencoder with just two nodes on the latent representation, we can plot what latent representation each input is assigned. If the autoencoder works well, we expect to see similar images clustered together (for instance smiling people vs frowning people, men vs women, etc).

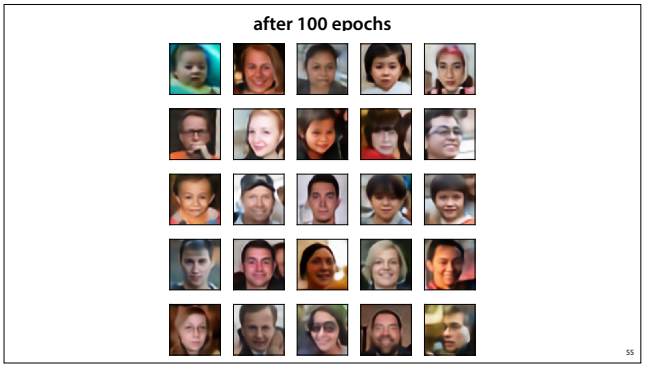
In a 2D space, we can't cluster too many attributes together, but in higher dimensions it's easier. To quote **Geoff Hinton**: "If there was a 30 dimensional supermarket, [the anchovies] could be close to the

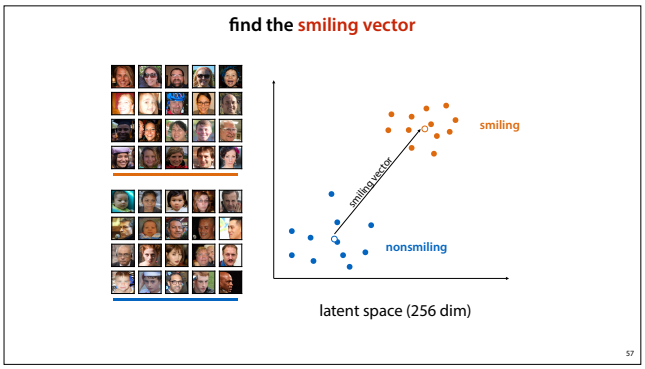
pizza toppings and close to the sardines."



Here are the reconstructions on a very simple network (just 4 fully connected ReLU layers for the encoder and the decoder), with MSE loss on the output.



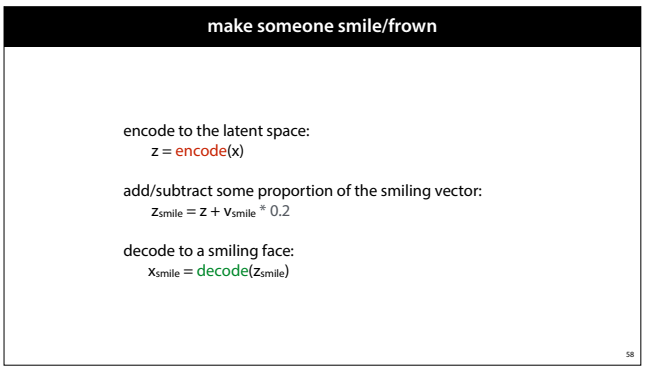


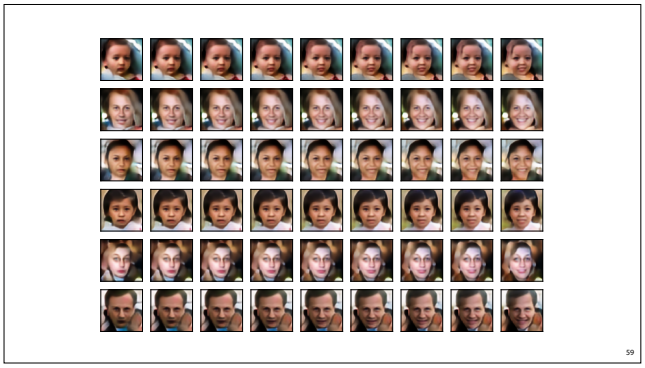


One thing we can now do is to study the latent space based on the exmaples that we have. For instance, we can see to smiling and non-smiling people end up in distinct parts of the latent space.

We just label a small amount of instances as smiling and nonsmiling. If we compute the means of the two resulting clusters in latent space, we can draw a vector between them. We can think of this as a "smiling" vector. The further we push people along this line, the more the decoded point will smile.

This is one big benefit of autoencoders: we can train them on unlabeled data (which is cheap) and then use only a *very* small number of labeled examples to





With a bit more powerful model, and some face detection, we can see what some famously moody celebrities might look like if they smiled.

---

autoencoders

keep the encoder and decoder: data manipulator.

keep the encoder, ditch the decoder: dimensionality reduction.

ditch the encoder, keep the decoder: generator network.

61

If we keep the encoder and the decoder, we get a network that can help us manipulate data in this way.

If we keep just the encoder, we get a powerful dimensionality reduction method. We can use the latent space representation as the features for a model that does not scale well to many features (like a non-naive Bayesian classifier).

But this lecture was about generator networks. How do we get a generator out of a trained autoencoder? It turns out we can do this by keeping just the decoder.

---

turning an autoencoder into a generator

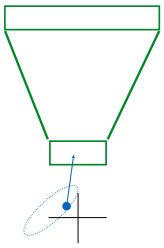
train an autoencoder

encode the data to latent variables Z

fit an MVN to Z

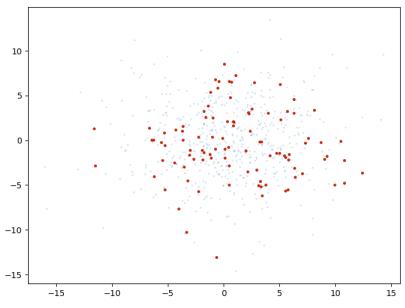
sample from the MVN

"decode" the sample



62

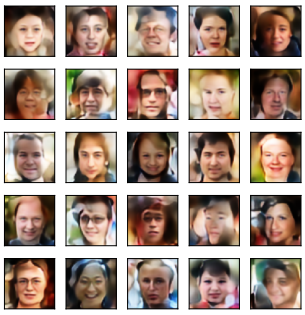
We don'tbeforehand know where the data will end up in latent space, but after training we can just check. We encode the training data, fit a distribution to this point cloud in our latent space, and then just use this distribution as the input to our decoder to create a generator network.



63

This is the point cloud of the latent representations in our example. We plot the first two of the 128 dimensions, resulting in the blue point cloud.

To these points we, we fit an MVN (in 128 dimensions), and we sample 400 points from it, the red dots.



generated

64

If we feed these points to the decoder, this is what we get.

How to control the shape of the latent space?

What are we optimizing? Can we optimize maximum likelihood directly?

Can we optimize for better interpolation directly?

65

This has given us a generator, but we have little control over what the latent space looks like. We just have to hope that it looks enough like a normal distribution that our MVN makes a good fit. In the GAN, we have perfect control over what our distribution on the latent space looks like; we can freely set it to anything. However, there, we have to fit a mapping from data to latent space after the fact.

We've also seen that this interpolation works well, but it's not something we've specifically trained the network to do. In the GAN, we should expect all latent space points to decode to something that fools the decoder, but in the autoencoder, there is nothing that stops the points in between the data points from decoding to garbage.

Moreover, neither the GAN nor the autoencoder is a very principled way of optimizing. Is there a way to train for **maximum likelihood** directly?

The answer to all of these questions is the **variational autoencoder**, which we'll discuss in the next video.

Deep generative models
Part 4: Variational autoencoders

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

variational autoencoders

- Force the decoder to also decode points *near* z correctly
- Forces the latent distribution of the data towards N(**0**, **I**)
- Can be derived from first principles
  maximum likelihood

67

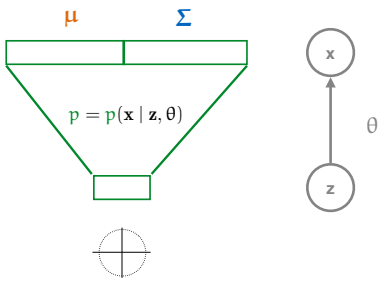
maximum (log) likelihood objective

$$\arg\max_{\theta} \quad \ln p(x \mid \theta)$$

68

We'll start with the maximum log-likelihood objective. We want to choose our parameters θ (the weights of the neural network) to maximise the log likelihood of the data. We will write this objective step by step until we end up with an autoencoder.

hidden variable model

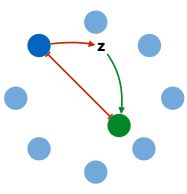


69

The first insight is that we can view our generator as a hidden variable model. We have a hidden variable **z**, which is standard normally distributed and then fed to a neural network to produce a variable **x**, which we observe.

the network computes the conditional distribution of **x** given **z**.

mode collapse



70

Here we see how the hidden variable problem causes our mode collapse. If we knew which **z** was supposed to produce which **x**, we could feed that z to the network compute the loss between the output and x and optimize by backpropagation and gradient descent.

The problem, as in the previous lecture, is that we don't have the complete data. We don't know the values of **z**, only the values of **x**.

The solution is to introduce

introducing q

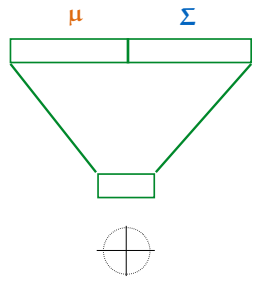
$$q(\mathbf{z} \mid \mathbf{x})$$

71

Our solution in the EM algorithm was to introduce a new distribution q, which tells us for a given **x**, which values of **z** are most likely. We'll follow the same strategy here. What would be a good q?

---

inverting p?



$$p(\mathbf{z} \mid \mathbf{x}, \theta)$$

72

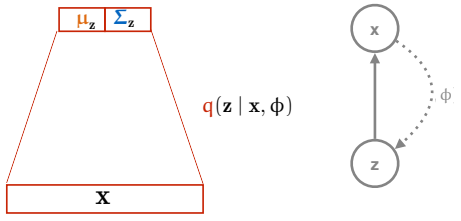
In the EM algorithm,for a given choice of parameters for p, we coudl easily work out the distribtion on **z**, conditioned on **x**. This gave us a good choice for q.

Here, things aren't so easy. To work out p(z|x, theta) we would need to **invert** the neural network: work out for a particular output x, which input values z are likely to have caused that output.

This is not impossible to do (we saw something similar in at the start of the lecture), but its a costly and imprecise business. Just like we did with the GANs, it's best to introduce a network that will learn the inversion for us.

---

introducing q



approximation for $p(\mathbf{z} \mid \mathbf{x}, \theta)$

73

This is the network we'll use. It consumes **x** and it produces a normal distribution on **z**. It has its own parameters, phi, which are not related to the parameters theta of p.

We'll now figure out a way to train p and q in concert. We'll try to update the parameters of p to fit the data, and try we'll update the parameters of q to keep it a good approximation to the inversion of p.

---

simplify our notation

$$p(\mathbf{x} \mid \mathbf{z}, \theta) = p_{\mathbf{w}}(\mathbf{x} \mid \mathbf{z})$$
$$q(\mathbf{z} \mid \mathbf{x}, \phi) = q_{\mathbf{v}}(\mathbf{z} \mid \mathbf{x})$$

$$p(\mathbf{z} \mid \theta) = \mathcal{N}(\mathbf{z} \mid \mathbf{0}, \mathbf{I})$$

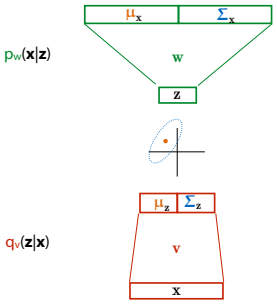
74

Since the parameters of our model are the neural network weights, we'll simplify our notation like this.

This empasizes that even though these are probabilities, the conditional probabilities shown here are the only probability densities that we can efficiently compute. We can't reverse the conditional, or maginalize anything out. The price we pay for using the power of neural networks is that we are stuck with these functions and have build an algorithm on just these.

With one exception. We do know the marginal distribution on **z**, since that is what we defined as the distribution on the input of our generator neural net:

it's a simple standard normal MVN.



75

Putting everything together, this is our model. If we feed q_v an instance from our data **x**, we get a normal distribution on the latent space. If we sample a point **z** from this distribution, and feed it to p_w we get a distribution on **x**. If the networks are both well trained, this should give us a good reconstruction of **x**.

The neural network p_w is our probability distribution conditional on the latent vector. q_v is our approximation of the conditional distribution on **z**.

a very useful decomposition

$$\ln p(\mathbf{x} \mid \theta) = L(q, \theta) + KL(q, p)$$

with :

$$p = p(\mathbf{z} \mid \mathbf{x}, \theta)$$

$q(\mathbf{z} \mid \mathbf{x})$ any approximation to $p(\mathbf{z} \mid \mathbf{x})$

$KL(q, p)$ Kullback-Leibler divergence

$$L(q, \theta) = \mathbb{E}_q \ln \frac{p(\mathbf{x}, \mathbf{z} \mid \theta)}{q(\mathbf{z} \mid \mathbf{x})}$$

76

We can now apply the decomposition we proved in the last lecture. We look at the marginal distribution on x, and break it up into

Before, we had a discrete hidden variable, and now we have a continuous one, but the proof still works (since we wrote everything in terms of expectations).

a very useful decomposition

$$\arg\max_w \sum_x \ln p_w(\mathbf{x})$$

$$\ln p_w(\mathbf{x}) = L(q_v, p_w) + KL(q_v, p_w)$$

with:

$q_v(\mathbf{z} \mid \mathbf{x})$: any approximation to $p_w(\mathbf{z} \mid \mathbf{x})$

$$L(q_v, p_w) = \mathbb{E}_q \ln \frac{p_w(\mathbf{x}, \mathbf{z})}{q_v(\mathbf{z} \mid \mathbf{x})}$$

77

Here is the decomposition, rewritten in our new notation.

In the EM algorithm, we used this to set up an alternating optimization algorithm: first optimizing one term and then the other. To do that here, we'd need to minimize the KL divergence between q and the inverse of p. This is possible, but expensive. Instead, we'll follow a cheaper track.

what's our loss?

$$\ln p_w(\mathbf{x}) = L(q_v, p_w) + KL(q_v, p_w)$$

variational lower bound
or evidence lower bound (ELBO)

$$L(q_v, p_w) = \mathbb{E}_q \ln \frac{p_w(\mathbf{x}, \mathbf{z})}{q_v(\mathbf{z} \mid \mathbf{x})}$$

78

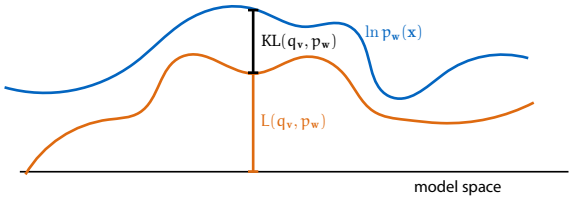
Because we cannot invert p_w, we cannot easily compute the KL term (let alone optimise q_v to minimise it).

Instead, we focus entirely on the L term. Since it's a lowerbound on the quantity we're trying to maximize, anything that increases L will help our model. The better we maximise L, the better our model will do.

---

lower bound loss

$$\ln p_w(\mathbf{x}) = L(q_v, p_w) + KL(q_v, p_w)$$



79

Here's a visualisation of how a lower bound objective works. We're interested in finding the highest point of the blue line (the maximum likelihood solution), but that's difficult to compute. Instead, we maximise the orange line (the evidence lower bound). Because it's guaranteed to be below the blue line everywhere, we may expect to be finding a high value for the blue line as well. To some extent, pushing up the orange line, pushes up the blue line as well.

How well we do on the blue line depends a lot on how *tight* the lower bound is. The distance between the lower bound and the log likelihood is expressed by the KL divergence between $p_w(z|x)$ and $q_v(z|x)$. That is, because we cannot easily compute $p_w(z|x)$, we introduced an approximation $q_v(z|x)$. The better this approximation, the lower the KL divergence, and the tighter the lower bound.

---

minimize -L(**v**, **w**)

$$-L(\mathbf{v}, \mathbf{w}) = -\mathbb{E}_q \ln \frac{p_w(\mathbf{x}, \mathbf{z})}{q_v(\mathbf{z} \mid \mathbf{x})}$$

$$= -\mathbb{E}_q \ln \frac{p_w(\mathbf{x} \mid \mathbf{z}) p_w(\mathbf{z})}{q_v(\mathbf{z} \mid \mathbf{x})}$$

$$= -\mathbb{E}_q \ln p_w(\mathbf{x} \mid \mathbf{z}) - \mathbb{E}_q \ln p_w(\mathbf{z}) + \mathbb{E}_q \ln q_v(\mathbf{z} \mid \mathbf{x})$$

$$= KL(q_v(\mathbf{z} \mid \mathbf{x}), p_w(\mathbf{z})) - \mathbb{E}_q \ln p_w(\mathbf{x} \mid \mathbf{z})$$

$$p_w(\mathbf{z}) = N(\mathbf{0}, \mathbf{I})$$

$$= KL(q_v(\mathbf{z} \mid \mathbf{x}), N(\mathbf{0}, \mathbf{I})) - \mathbb{E}_q \ln p_w(\mathbf{x} \mid \mathbf{z})$$

80

Right now, we can't use L as a loss function directly, since it contains functions like p(x,z) that are not easily computed and because it's an expectation. We'll rewrite it step by step until it's a loss function that can de directly used, heaply and easily in a deep learning system.

Since we want to implement a loss function, we will *minimize* the negative of the L function. We now need to work out what that means for our neural net.

All three probability functions are known: q(z|x) is the encoder network, p(x|z) is the decoder network, and p(z) was chosen when we defined (back in slide 9) how the generator works, if we ignore x, the distribution on

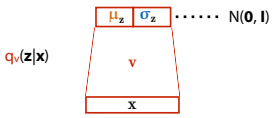
z is a standard multivariate normal.

NB: Since we now take the expectation over a continuous distribution, all sums become integrals and (inside all expectations and inside the KL divergence). By keeping everything written as expectations, we ensure that we don't have to deal with integrals.

---

loss function

$$\text{loss}(v, w) = \text{KL}(q_v(\mathbf{z} \mid \mathbf{x}), p_w(\mathbf{z})) - \mathbb{E}_q \ln p_w(\mathbf{x} \mid \mathbf{z})$$

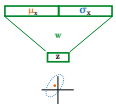


81

The KL term is just the KL divergence between the MVN that the encoder produce for x and the standard normal MVN. This **works out** as a relatively simple differentiable function of mu and sigma, so we can use it directly in a loss function.

Remember that Sigma_z is usually restricted to a diagonal matrix, so the network just outputs a vector of the same size as mu_z, which we take to be the diagonal of the covariance matrix.

$$\text{loss}(v, w) = \text{KL}(q_v(\mathbf{z} \mid \mathbf{x}), p_w(\mathbf{z})) - \mathbb{E}_q \ln p_w(\mathbf{x} \mid \mathbf{z})$$



take L samples $\{\mathbf{z}_i\}$ from $q(\mathbf{z}|\mathbf{x})$.

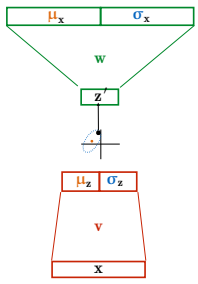
approximate as: $\frac{1}{L} \sum_i \ln p_w(\mathbf{x} \mid \mathbf{z}_i)$

keep things simple: L=1

82

The second part of our loss function requires a little more work. It's an **expectation** for which we don't have a closed form expression. Instead, we can *approximate* it by taking some samples, and averaging. To keep things simple, we just take a single sample (we'll be computing the network lots of times during training, so overall, we'll be taking lots of samples).

$$\text{loss}(\mathbf{v}, \mathbf{w}) = \text{KL}(q_v(\mathbf{z} \mid \mathbf{x}), \text{N}(\mathbf{0}, \mathbf{I})) - \ln p_w(\mathbf{x} \mid \mathbf{z}')$$



83

We almost have a fully differentiable model. Unfortunately, we still have a sampling step in the middle (and sampling is not a differentiable operation).

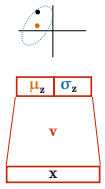
sampling

$N(\mu, \Sigma)$: $X\sigma + \mu$ with $X \sim N(0,1)$

$N^d(\mathbf{0}, \mathbf{1})$: $\begin{pmatrix} X_1 \\ \vdots \\ X_d \end{pmatrix}$ with $X_i \sim N(0,1)$

$N^d(\mu, \Sigma)$: $AX + \mu$ with $X \sim N^d(\mathbf{0}, \mathbf{1})$,
$\Sigma = AA^T$

We can get get rid of it by remembering the algorithm for sampling from a given MVN. We sample from the standard MVN, and *transform* the sample using the parameters of the required MVN.

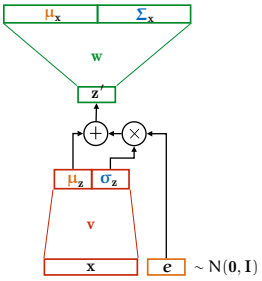


$$X \sim N^d(\mathbf{0}, \mathbf{I})$$

$$N^d(\mu, \Sigma) : X \otimes \sigma + \mu$$

85

$$\text{loss}(\mathbf{v}, \mathbf{w}) = \text{KL}(q_{\mathbf{v}}(\mathbf{z} \mid \mathbf{x}), N(\mathbf{0}, \mathbf{I})) - \ln p_{\mathbf{w}}(\mathbf{x} \mid \boldsymbol{e} \otimes \sigma_{\mathbf{z}} + \mu_{\mathbf{z}})$$

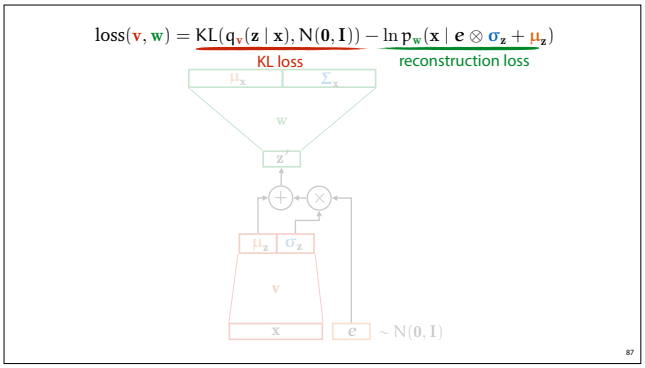


86

We can work this sampling into the architecture. We provide the network with an **extra input**: a sample from the standard MVN. We also slightly change the q network.

Instead of making the network produce Sigma, we make it produce A, since the network is trained anyway, we can just interpret the output as A instead of Sigma. Since both are diagonal matrices A just contains the square roots of Sigma.

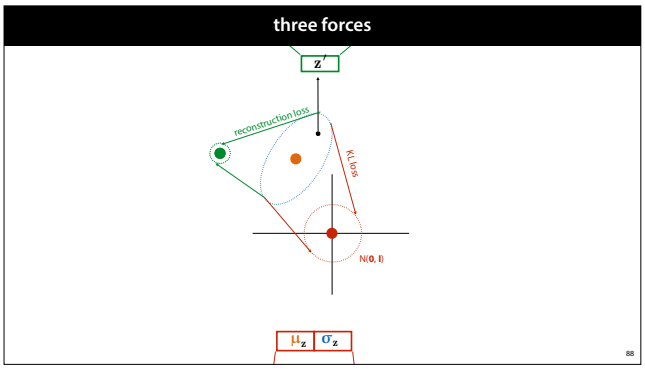
Why does this help us? We're still sampling, but we've moved the random sampling out of the way of the backpropagation. The gradient can now propagate down to the weights of the q function, and the actual randomness is treated as an *input*, rather than a computation.



The two terms of the loss function are usually called KL loss and reconstruction loss.

The reconstruction loss maximises the probability of the current instances. This is basically the same loss we used for the regular auto-encoder: we want the output of the decoder to look like the input.

The KL loss ensures that the latent distributions are clustered around the origin, with variance 1. Over the whole dataset, it ensures that the latent distribution looks like a standard normal distribution.

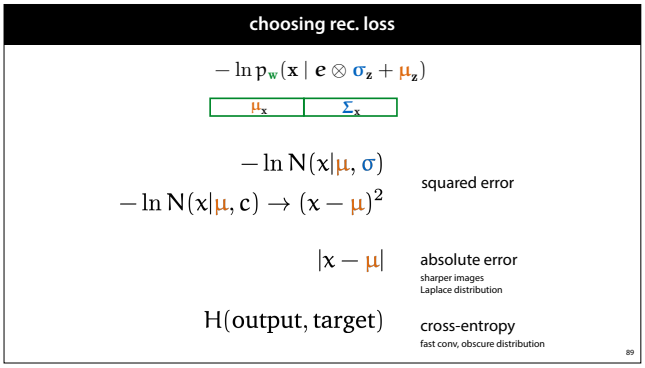


The formulation of the VAE has three forces acting on the latent space. The reconstruction loss pulls the latent distribution as much as possible towards a single point that gives the best reconstruction. Meanwhile, the KL loss, pulls the latent distribution (for all points) towards the standard normal distribution, acting as a **regularizer**. Finally, the sampling step ensures that not just a single point returns a good reconstruction, but a whole *neighbourhood* of points does. The effect can be summarized as follows:

The **reconstruction loss** ensures that there are points in the latent space that decode to the data.

The **KL loss** ensures that all these points together are laid out like a standard normal distribution.

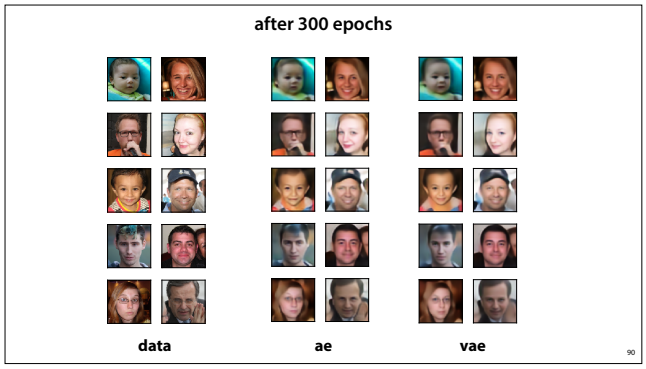
The **sampling step** ensure that points in between these also decode to points that resemble the data.

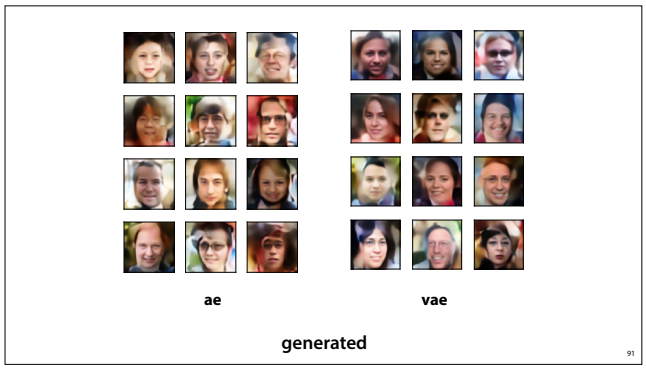


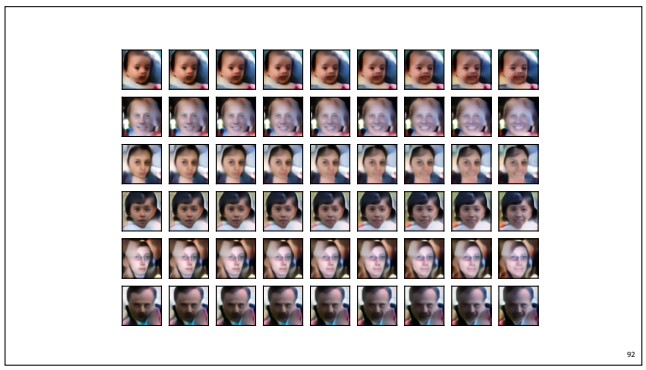
The two terms of the loss function are usually called KL loss and reconstruction loss.

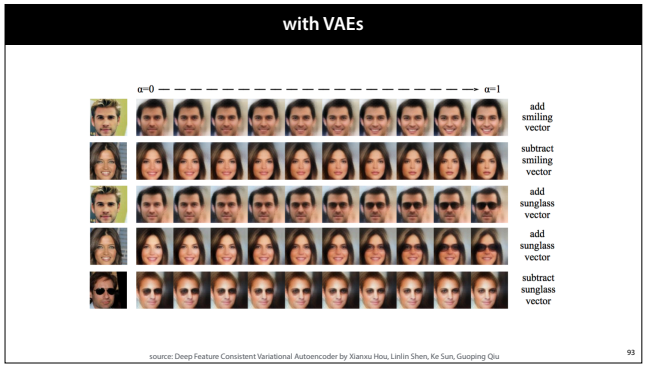
The reconstruction loss maximises the probability of the current instances. This is basically the same loss we used for the regular auto-encoder: we want the output of the decoder to look like the input.

The KL loss ensures that the latent distributions are clustered around the origin, with variance 1. Over the whole dataset, it ensures that the latent distribution looks like a standard normal distribution.





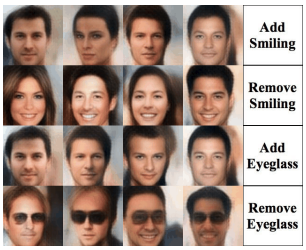




Here are some examples from a more elaborate VAE

source: Deep Feature Consistent Variational Autoencoder by Xianxu Hou, Linlin Shen, Ke Sun, Guoping Qiu

with VAEs



source: https://houxianxu.github.io/assets/project/dfcvae

94

source: **https://houxianxu.github.io/assets/project/dfcvae**

---

from worksheet 5

```
for epoch in range(5):
    for images, _ in tqdm(trainloader): # if tqdm gives you trouble just remove it
        b, c, h, w = images.size()

        # forward pass
        z = encoder(images)

        # - split z into mean and sigma
        zmean, zsig = z[:, :latent_size], z[:, latent_size:]
        kl = kl_loss(zmean, zsig)

        zsample = sample(zmean, zsig)

        o = decoder(zsample)
        rec = F.binary_cross_entropy(o, images, reduction='none')
        rec = rec.view(b, c*h*w).sum(dim=1)
        # -- Reconstruction loss. We ask pytorch not to sum the loss, and sum over the
        #    channels and pixels ourselves. This gives us a loss per instance that we
        #    can add to the kl loss

        loss = (rec + kl).mean() # sum the losses and take the mean
        loss.backward()
```

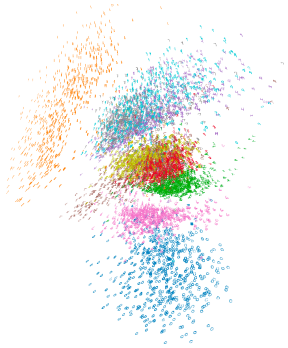
https://github.com/mlvu/worksheets/blob/master/Worksheet%205%2C%20Pytorch.ipynb

95

Here is what the algorithm looks like in pytorch. Load the 5th worksheet to give it a try.

https://github.com/mlvu/worksheets/blob/master/Worksheet%205%2C%20Pytorch.ipynb

---



96

In this worksheet, the VAE is trained on MNIST data, with a 2D latent space. Here is the original data, plotted by their latents coordinates. The colors represent classes, to which the VAE did not have access.

If you ruen the worksheet, you'll end up with this picture.

---

interpolation

AE

i went to the store to buy some groceries .
i store to buy some groceries .
i were to buy any groceries .
horses are to buy any groceries .
horses are to buy any animal .
horses the favorite any animal .
horses the favorite favorite animal .
horses are my favorite animal .

VAE

he was silent for a long moment .
he was silent for a moment .
it was quiet for a moment .
it was dark and cold .
there was a pause .
it was my turn .

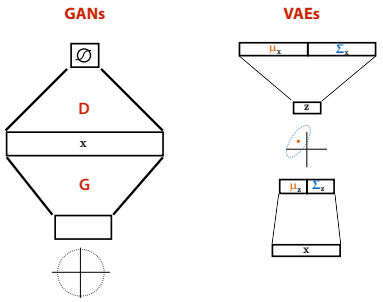
97

While the added value of the VAE is a bit difficult to detect in our example, in other domains it's more clear.

Here is an example of interpolation on sentences. First using a regular autoencoder, and then using a VAE. Note that the intermediate sentences for the AE are non-grammatical, but the intermediate sentences for the VAE are all grammatical.

source: Generating Sentences from a Continuous Space by Samuel R. Bowman, Luke Vilnis, Oriol Vinyals, Andrew M. Dai, Rafal Jozefowicz, Samy Bengio https://arxiv.org/abs/1511.06349

summary: generative modeling



98

We see that GANs are in many ways the inverse of autoencoders, in that GANS have the data space as the inside of the network, and VAEs have it as the outside.

GANs and VAEs

Allow us to train generator networks, avoiding mode collapse

GANs	VAEs
Better for images, often poor in other domains.	Work for language, music, etc.
Ad-hoc model, difficult to establish what is being optimized.	Derived from first principles.
Can't handle discrete data easily.	Allow mapping from data to latent space.
Derived by *inverting* a discriminator.	Can't handle discrete latent variables easily.
	Derived by *inverting* a generator.

99

VAEs and PCA

Mapping to low-dimensional *whitened* latent space (zero, mean, decorrelated).

PCA	VAEs
Linear transformation	Nonlinear transformation
Analytical solution	GD required
Principal components often meaningful, ordered by impact.	Latent dimensions not usually meaningful. *Directions* in latent space meaningful.

100

---

mlcourse@peterbloem.nl